ACM321 Java Lab9

Polymorphism in Java

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Polymorphism allows us to perform a single action in different ways.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

1. **Compile time polymorphism:**

   It can also be called **Static polymorphism**, this type of polymorphism is achieved by **function overloading.**

   Method Overloading

   When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.

   Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments.**

```
public class Multiplier {
  public int multiply(int a, int b) {
    return a * b;
  }

//different types of arguments
  public double multiply(double a, double b) {
    return a * b;
  }

//different number of arguments
  public int multiply(int a, int b, int c) {
    return a * b * c;
  }
}
```

```
public class ProgramMain {

  public static void main(String[] args) {
    Multiplier m1 = new Multiplier();

    System.out.println(m1.multiply(3, 5));
    System.out.println(m1.multiply(3.2, 5.7));
    System.out.println(m1.multiply(3, 5, 8));
  }
```

```
}
```

2. Runtime Polymorphism (Dynamic Method Dispatch)

It is a process in which a function call to the overridden method is resolved at Runtime.

This type of polymorphism is achieved by Method Overriding.

> Lets remember method overriding from inheritance;
> **Method overriding**, occurs when a **derived class** has a definition for one of the member functions of the **base class**. That **base function** is said to be **overridden**.

```
public class ProgramMain {

  public static void main(String[] args) {

        A a = new A();

        A b = new B();

        A c = new C();

        a.method1();
        b.method1();
        c.method1();

    /*_________Output will be
    * Inside A's m1 method
    * Inside B's m1 method
    * Inside C's m1 method
    */
  }

}
```

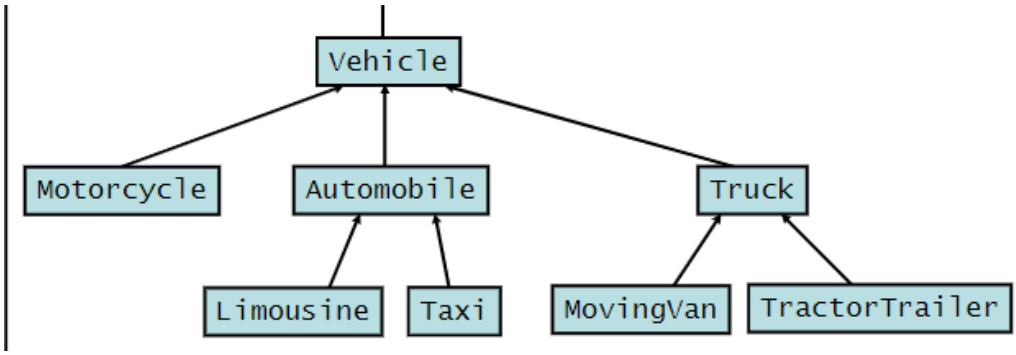
```
//class A
public class A {
  void method1() {
    System.out.println("Inside A's m1 method");
  }
}
```

```
//class B
public class B extends A {
  void method1() {
    System.out.println("Inside B's m1 method");
  }
}
```

```
//class C
public class C extends A {
  void method1() {
    System.out.println("Inside C's m1 method");
  }
}
```

Dynamic Binding

Lets see a real life analogy; Vehicle and derived classes from it.



Declared Type vs. Actual Type

- An object's declared type may not match its actual type:
  - declared type: type specified when declaring a variable
  - actual type: type specified when creating an object
- Recall this client code:

```
Vehicle[] fleet = new Vehicle[5];
fleet[0] = new Automobile("Honda", "Civic", 2005);
fleet[1] = new Motorcycle("Harley", …);
fleet[2] = new TractorTrailer("Mack", …);
```

- Here are the types:

object	declared type	actual type
fleet[0]	Vehicle	Automobile
fleet[1]	Vehicle	Motorcycle
fleet[2]	Vehicle	TractorTrailer

```
//Vehicle class
public class Vehicle {
  public String brand;
  public String model;

  public void info() {
    System.out.println("vehicle info");
  }
}
```

```
//Automobile class
public class Automobile extends Vehicle{
  public void info() {
    System.out.println("automobile info");
  }
}
```

```
public class Truck extends Vehicle{
  public void info() {
    System.out.println("truck info");
  }
}
```

```
public class Motocycle extends Vehicle{
  //no overriding method
}
```

```
//ProgramMain.java

public class ProgramMain {

  public static void main(String[] args) {
    //fleet : filo [TR]
    Vehicle[] fleet = new Vehicle[5];

    fleet[0] = new Automobile();
    fleet[1] = new Truck();
    fleet[2] = new Motocycle();

    fleet[0].info();
    fleet[1].info();
    fleet[2].info();
  }

}
```

Objects represented by a
declared type of **Vehicle**
and **actual type** of **Automobile, Truck, Motocycle**

Which version will be executed?
How does the interpreter decide which version of a method should be used?

At runtime, the Java interpreter selects the version of a method that is appropriate to the actual type of the object.
starts at the actual type, and goes up the inheritance hierarchy as needed until it finds a version of the method, known as **dynamic binding**.

Some advantages of polimorphism

Polymorphism allows us to easily write code that works for more than one type of object.

Without polymorphism, we would need a large if-else-if:

```
if (fleet[i] is an Automobile)
  { print the appropriate info for Automobiles }
else if (fleet[i] is a Truck)
  { print the appropriate info for Trucks }
else if ...
```

Extra info for runtime polymorphism

Runtime Polymorphism with Data Members

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.** For example :

```java
// Java program to illustrate the fact that
// runtime polymorphism cannot be achieved
// by data members

// class A
class A
{
    int x = 10;
}

// class B
class B extends A
{
    int x = 20;
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

```
10
```

Explanation : In above program, both the class A(super class) and B(sub class) have a common variable 'x'. Now we make object of class B, referred by 'a' which is of type of class A. Since variables are not overridden, so the statement "a.x" will **always** refer to data member of super class.

Refs and Other Resources

Polymorphism in Java - GeeksforGeeks

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Real life example of polymorphism: A person at the same time can have

https://www.geeksforgeeks.org/polymorphism-in-java/

Dynamic Method Dispatch or Runtime Polymorphism in Java - GeeksforGeeks

Prerequisite: Overriding in java, Inheritance Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java